Lambda calcolo

APhEx 16, 2017 (ed. Vera Tripodi)
Ricevuto il: 15/05/2017
Accettato il: 15/10/2017
Redattore: Vera Tripodi & Pierluigi Graziani

APhEx
PORTALE ITALIANO DI FILOSOFIA ANALITICA
GIORNALE DI FILOSOFIA
NETWORK
N°16 GIUGNO 2017

T E M I

LAMBDA CALCOLO

di Mariangiola Dezani-Ciancaglini e Betti Venneri

ABSTRACT – Lo scopo di questo tema è quello di fornire un'introduzione informale al lambda calcolo con squarci su alcune delle molte influenze che esso ha avuto nello sviluppo recente dell'informatica. Il desiderio è quello di sollecitare la curiosità del lettore verso approfondimenti che, nel migliore dei casi, potrebbero portarlo a svolgere ricerche in un campo ancora ricco di possibilità inesplorate.

1. INTRODUZIONE

La nozione prevalente di funzione nella matematica moderna è quella di sottoinsieme del prodotto cartesiano di due insiemi. Quindi, scrivendo $f : X \to Y$ intendiamo una funzione che associa ad ogni valore di X un determinato valore di Y, per qualunque coppia di insiemi X e Y detti, rispettivamente, dominio e condominio. In questa visione *estensionale* delle funzioni, non importa quale sia, neanche se esista, il procedimento di calcolo che porta dall'argomento (in inglese "input") in X al risultato (in inglese "output") in Y. L'associazione fra valori del dominio e del condominio è l'unico elemento che caratterizza una funzione, le sue proprietà, l'uguaglianza con un'altra funzione. Completamente opposto a questo approccio è la visione *intensionale* delle funzioni, in cui ogni funzione è caratterizzata dalla descrizione delle regole di calcolo che permettono di arrivare dall'argomento al valore da restituire come risultato. Per esempio, consideriamo due semplici varianti della funzione successore:

$$\mathsf{succ}(x) = x + 1 \qquad\qquad \mathsf{succLento}(x) = (x - 2) + 1 + 1 + 1$$

Queste due funzioni sono intensionalmente diverse, anche se hanno esattamente lo stesso insieme di coppie di numeri naturali. Possiamo confrontarle per concludere che la prima sia preferibile alla seconda perché il suo procedimento di calcolo è più veloce.

In questo approccio intensionale si inserisce il *lambda calcolo*, disegnato da *Alonso Church* (cfr. Church (1932)) con lo scopo di ottenere una fondazione della matematica mediante un sistema di funzioni ed un insieme di nozioni logiche. La lambda notazione semplifica

la scrittura delle funzioni attraverso l'uso del simbolo λ, che introduce il parametro della funzione. L'espressione $\lambda x.x + 1$ è un esempio di *lambda termine*: è chiamato *lambda astrazione* perché inizia con una λ. Notiamo che $\lambda x.x + 1$ descrive la funzione succ sopra definita, mentre succLento è descritta da un altro lambda termine, precisamente $\lambda x.(x - 2) + 1 + 1 + 1$.

Nel *lambda calcolo* la valutazione della funzione succ per l'argomento 2 diventa:

$$(\lambda x.x + 1)2 \longrightarrow 2 + 1 \longrightarrow 3$$

Il lambda termine $(\lambda x.x + 1)2$ è chiamato *applicazione*: rappresenta l'applicazione della funzione $\lambda x.x + 1$ all'argomento 2. La freccia

$$(\lambda x.x + 1)2 \longrightarrow 2 + 1$$

denota l'operazione di sostituzione della variabile x con 2 nel lambda termine $x + 1$. È un esempio di uso dell'unica regola di valutazione (o riduzione) dei lambda termini: la *β-regola* (in inglese "β-rule").

È interessante notare come la lambda notazione naturalmente trasformi le funzioni di più argomenti in funzioni di un solo argomento che restituiscono funzioni. Ad esempio, la funzione somma nella lambda notazione diventa:

$$\mathsf{somma} = \lambda y.\lambda x.x + y$$

e la sua applicazione ad 1 si riduce alla funzione succ:

$$\mathsf{somma}\,1 = (\lambda y.\lambda x.x + y)\,1 \longrightarrow \lambda x.x + 1 = \mathsf{succ}$$

Questo fenomeno prende il nome di *currificazione* delle funzioni, in omaggio ad *Haskell Curry*, che è con Alonzo Church uno dei fondatori del lambda calcolo (cfr. Curry (1934)). Storicamente, però, la nozione di currificazione è stata introdotta da Moses Schönfinkel (cfr. Schönfinkel (1924)).

Come si vede, il risultato dell'applicazione di somma ad 1 è succ, cioè è, a sua volta, una funzione. D'altra parte una funzione può essere passata come argomento ad un'altra funzione, ad esempio possiamo applicare $\lambda y.y2$ a $\lambda x.x + 1$:

$$(\lambda y.y2)(\lambda x.x + 1) \longrightarrow (\lambda x.x + 1)2 \longrightarrow 2 + 1 \longrightarrow 3$$

La possibilità di avere funzioni come argomenti e risultati di funzioni è un pregio fondamentale del lambda calcolo: queste funzioni sono chiamate *funzioni di ordine superiore* (in inglese "higher-order functions").

La lambda notazione permette di usare le variabili sia per rappresentare gli argomenti delle funzioni che le costanti. Ad esempio, il lambda termine

$$\lambda x.x - y$$

è la funzione che applicata ad un argomento gli sottrae la costante y, mentre il lambda

termine

$$\lambda y.x - y$$

è la funzione che applicata ad un argomento lo sottrae alla costante x. Questa distinzione è dovuta all'operatore λ, che *lega* (in inglese "bounds") la variabile a cui è applicato, così come in logica il quantificatore universale (operatore "per ogni", notazione $\forall$) lega le variabili. Un esempio di formula logica quantificata universalmente è:

$$\forall x.x + 1 > x$$

che si legge: "per ogni x la somma di x con 1 è maggiore di x", sottointendendo che x sia un numero.

Una variable non legata si dice *libera*. Ritornando agli esempi precedenti:

- la variabile x è legata in $\lambda x.x + 1$, $\lambda x.\lambda y.x + y$, $\lambda x.x - y$, ma è libera in $\lambda y.x - y$,
- la variabile y è legata in $\lambda x.\lambda y.x + y$, $\lambda y.x - y$, $\lambda y.y2$, ma è libera in $\lambda x.x - y$.

In realtà l'essere libera o legata non è una proprietà di una variabile, ma di un'occorrenza di una variabile. Questo è vero anche per le formule logiche. Ad esempio consideriamo

$$x\&(\forall x.x \vee \neg x)$$

dove $\&$, $\vee$, $\neg$ sono rispettivamente gli operatori logici di congiunzione, disgiunzione e negazione. In questa formula tutte le occorrenze di x rappresentano valori booleani, ma l'oc-

correnza più a sinistra di x è libera, mentre le altre due occorrenze di x sono quantificate universalmente da $\forall$.

Avere occorrenze libere e legate della stessa variabile o peggio occorrenze legate da diverse occorrenze di operatori rende poco leggibili le formule logiche ed i lambda termini. Inoltre la riduzione dei lambda termini può diventare complessa ed è stata argomento di un'ampia ricerca. Fortunatamente si può osservare che i nomi delle variabili legate non sono significativi. Ad esempio, la funzione succ possiamo definirla come $\lambda x.x + 1$ oppure $\lambda y.y + 1$ oppure $\lambda z.z + 1$ etc. Per questo motivo *Henk Barendregt* (cfr. Barendregt (1984) (Convenzione 2.1.12)) ha proposto che i nomi delle variabili legate da diverse occorrenze di operatori siano sempre diversi fra di loro e diversi dai nomi delle variabili libere. L'esempio precedente deve quindi essere riscritto come $x\&(\forall y.y \vee \neg y)$ oppure $x\&(\forall z.z \vee \neg z)$ etc.

L'α-equivalenza (notazione $\equiv$) fra lambda termini è definita proprio mediante la rideno-mina delle variabili legate, ad esempio $\lambda x.\lambda y.x + y \equiv \lambda z.\lambda t.z + t$, ma $\lambda x.\lambda y.x + y \not\equiv \lambda z.\lambda t.t + z$ e $\lambda x.\lambda y.x + y \not\equiv \lambda z.\lambda t.z + z$.

Per semplicità in queste note useremo la convenzione di Barendregt e considereremo i lambda termini modulo α-equivalenza.

Nota storica Troviamo le nozioni di astrazione e sostituzione già negli assiomi per l'aritmetica di Giuseppe Peano (cfr. Peano (1889)), però senza l'idea di valutazione. Moses Schönfinkel ha creato la Logica Combinatoria (cfr. Schönfinkel (1924)), un formalismo che differisce dal lambda calcolo essenzialmente per l'eliminazione delle variabili legate. In questo lavoro la nozione di equivalenza fra termini della Logica Combinatoria è

usata informalmente: essa è stata definita nella tesi di Haskell Curry (cfr. Curry (1930)) usando la riduzione. Scopo di Curry era la creazione di un sistema formale per studiare approfonditamente le nozioni logiche nel contesto più ampio possibile.

2. SINTASSI E SEMANTICA

Nella sezione precedente a scopo illustrativo abbiamo considerato i lambda termini costruiti a partire da naturali ed operazioni sui naturali. Come definizione formale di lambda calcolo usiamo, invece, la seguente, che non include costanti. Nella letteratura ci sono varie definizioni e il presente calcolo, che è il più semplice, prende il nome di *lambda calcolo puro*. Vedremo nella sezione 3. che i naturali e le operazioni sui naturali si possono rappresentare nel lambda calcolo puro.

Definizione 1 (Lambda Termini)**.** *I termini del lambda calcolo, o lambda termini, sono costruiti a partire da un insieme infinito enumerabile di variabili* x, y, z, t, u, v, x_1, y_1, z_1, t_1, u_1, v_1, x_2, y_2, z_2, t_2, u_2, v_2, ... *nel modo seguente:*

- *ogni variabile è un lambda termine;*
- *se* M, N *sono lambda termini, allora anche la loro applicazione* (MN) *è un lambda termine;*
- *se* M *è un lambda termine, allora la lambda astrazione* $(\lambda x.M)$ *è un lambda termine.*

Nello scrivere i lambda termini si usa minimizzare il numero delle parentesi secondo le seguenti convenzioni:

- l'applicazione associa a sinistra, ad esempio MNP rappresenta $((MN)P)$;

- l'astrazione associa a destra, ad esempio $\lambda x.\lambda y.M$ rappresenta $(\lambda x.(\lambda y.M))$.

Semplici lambda termini sono:

- la funzione identità $\lambda x.x$ (usualmente denotata da I);
- la funzione che applicata a due argomenti sceglie il primo $\lambda x.\lambda y.x$ (usualmente denotata da K);
- la funzione che applicata a due argomenti sceglie il secondo $\lambda x.\lambda y.y$ (usualmente denotata da $\overline{0}$).

Le definizioni di α-equivalenza e di β-regola richiedono la formalizzazione della *sostituzione* delle occorrenze di una variabile x in un lambda termine M con un lambda termine N, notazione $M[x := N]$. Senza condizioni su M ed N questa formalizzazione è alquanto complessa, si veda ad esempio Hindley e Seldin (2008) (Definizione 1.12). Assumendo la convenzione di Barendregt possiamo semplicemente definire la sostituzione $M[x := N]$ per induzione su M come segue:

$$\begin{aligned} x[x := N] &= N \\ y[x := N] &= y \text{ dove } y \neq x \\ (M_1 M_2)[x := N] &= M_1[x := N] M_2[x := N] \\ (\lambda y.M')[x := N] &= \lambda y.M'[x := N] \end{aligned}$$

Ad esempio:

$$(\lambda y.xy)[x := \lambda z.z] = \lambda y.(xy)[x := \lambda z.z] = \lambda y.x[x := \lambda z.z]y[x := \lambda z.z] = \lambda y.(\lambda z.z)y$$

Un ulteriore ingrediente necessario è la nozione di sottotermine di un lambda termine, che si definisce facilmente per casi:

- M è un sottotermine di M;
- se M è un sottotermine di N, allora M è un sottotermine di NN', $N'N$ e $\lambda x.N$.

Ad esempio, i sottotermini di $\lambda y.(\lambda z.z)y$ sono $\lambda y.(\lambda z.z)y$, $(\lambda z.z)y$, $\lambda z.z$, y, z.

Ora possiamo definire formalmente sia l'α-equivalenza (generata dall'α-regola) che la β-regola (che induce la β-riduzione), entrambe esemplificate nella sezione precedente.

Definizione 2 (α-regola e α-equivalenza)**.** *L'α-regola eguaglia due lambda termini che differiscono per il nome di una variabile legata:*

$$(\alpha) \qquad \lambda x.M =_\alpha \lambda y.M[x := y]$$

L'α-equivalenza (notazione $\equiv$) eguaglia due lambda termini che si possono ottenere l'uno dall'altro mediante un numero finito (anche zero) di applicazioni dell'α-regola a sottotermini. In altre parole $\equiv$ è la relazione di congruenza indotta dalla chiusura riflessiva, simmetrica e transitiva di $=_\alpha$.

Definizione 3 (β-regola e β-riduzione)**.** *La β-regola riscrive l'applicazione di una lambda astrazione $\lambda x.M$ and un lambda termine N nella sostituzione di N alle occorrenze libere di x in M:*

$$(\beta) \qquad (\lambda x.M)N \longrightarrow_\beta M[x := N]$$

Il termine $(\lambda x.M)N$ *è chiamato* β-redesso *(in inglese "*β*-redex") ed il termine* $M[x := N]$ *è il suo* contratto *(in inglese "contractum").*

Il termine M *si* β*-riduce in un passo al termine* N *(notazione* $M \longrightarrow N$*) se* N *è ottenuto da* M *applicando la* β*-regola ad un sottotermine di* M*.*

Il termine M *si* β*-riduce al termine* N *(notazione* $M \longrightarrow^\star N$*) se* N *è ottenuto da* M *mediante un numero finito (anche zero) di applicazioni della* β*-riduzione in un passo.*

In altre parole $\longrightarrow$ *è la relazione di congruenza indotta da* $\longrightarrow_\beta$ *e* $\longrightarrow^\star$ *è la chiusura riflessiva e transitiva di* $\longrightarrow$*.*

Ad esempio:

$$\underline{(\lambda x.\lambda y.xy)(\lambda z.z)} \longrightarrow \lambda y.\underline{(\lambda z.z)y} \longrightarrow \lambda y.y$$

In questo e nei successivi esempi il β-redesso contratto in ogni passo è sottolineato.

Dato che la β-regola può sostituire lo stesso lambda termine a più occorrenze di una variable è necessario a volte utilizzare l'α-regola per ottenere un lambda termine che rispetti la convenzione di Barendregt. Questo è sempre lecito dato che consideriamo i lambda termini modulo α-equivalenza. Ad esempio:

$$\underline{(\lambda x.\lambda y.yxx)(\lambda z.z)} \longrightarrow \lambda y.y(\lambda z.z)(\lambda t.t)$$

In generale in un lambda termine ci sono più β-redessi, e per questo motivo la β-riduzione

non è unica, ad esempio:

$$\begin{aligned}
\underline{(\lambda x.xx)((\lambda y.y)(\lambda z.z))} &\longrightarrow \underline{(\lambda y.y)(\lambda z.z)}((\lambda t.t)(\lambda v.v)) \longrightarrow \underline{(\lambda z.z)((\lambda t.t)(\lambda v.v))} \\
&\longrightarrow \underline{(\lambda t.t)(\lambda v.v)} \longrightarrow \lambda v.v \\
(\lambda x.xx)(\underline{(\lambda y.y)(\lambda z.z)}) &\longrightarrow \underline{(\lambda x.xx)(\lambda z.z)} \longrightarrow \underline{(\lambda z.z)(\lambda v.v)} \\
&\longrightarrow \lambda v.v \\
\underline{(\lambda x.xx)((\lambda y.y)(\lambda z.z))} &\longrightarrow (\lambda y.y)(\lambda z.z)(\underline{(\lambda t.t)(\lambda v.v)}) \longrightarrow \underline{(\lambda y.y)(\lambda z.z)}(\lambda v.v) \\
&\longrightarrow \underline{(\lambda z.z)(\lambda v.v)} \longrightarrow \lambda v.v
\end{aligned}$$

Una *strategia di riduzione* è una scelta sistematica di un β-redesso da contrarre in un lambda termine arbitrario. La strategia *per nome* (in inglese "call-by-name") riduce il β-redesso più esterno e più a sinistra, la strategia *per valore* (in inglese "call-by-value") riduce il β-redesso più interno e più a sinistra. La prima riduzione dell'esempio precedente usa la strategia per nome, la seconda usa la strategia per valore e l'ultima sceglie i β-redessi senza seguire una strategia.

Un lambda termine che non contiene β-redessi è chiamato una *forma normale*, ad esempio $\lambda x.xx$, $\lambda y.y$ sono forme normali. Non tutti i lambda termini si riducono ad una forma normale, il più semplice esempio è $(\lambda x.xx)(\lambda y.yy)$ che si riduce sempre a se stesso:

$$\underline{(\lambda x.xx)(\lambda y.yy)} \longrightarrow \underline{(\lambda x.xx)(\lambda y.yy)} \longrightarrow \underline{(\lambda x.xx)(\lambda y.yy)} \longrightarrow \cdots$$

Un lambda termine può ridursi o meno ad una forma normale a seconda della strategia di

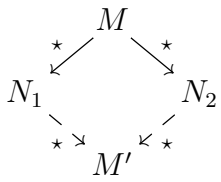


Figura 1: Teorema di Church-Rosser.

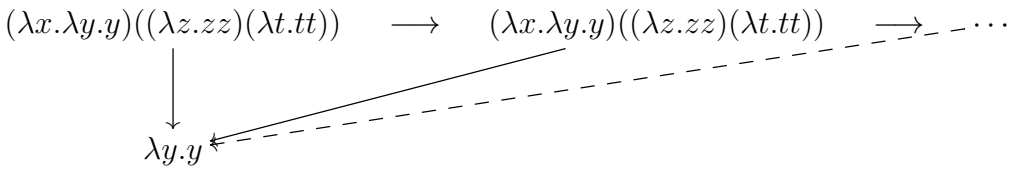


Figura 2: Esempio di applicazione del Teorema di Church-Rosser.

riduzione usata. Ad esempio, usando la strategia per nome:

$$\underline{(\lambda x.\lambda y.y)((\lambda z.zz)(\lambda t.tt))} \longrightarrow \lambda y.y$$

mentre usando la strategia per valore

$$(\lambda x.\lambda y.y)\underline{((\lambda z.zz)(\lambda t.tt))} \longrightarrow (\lambda x.\lambda y.y)\underline{((\lambda z.zz)(\lambda t.tt))} \longrightarrow \cdots$$

Il teorema di Church-Rosser, che certamente è una pietra miliare del lambda calcolo, ci assicura che se un lambda termine ha una forma normale questa è unica. Questo teorema dice che se riduciamo in due modi diversi lo stesso lambda termine M ottenendo i termini N_1 ed N_2, possiamo sempre trovare un lambda termine M' tale che entrambi N_1 ed N_2 si riducano ad M'. Chiaramente se M' è una forma normale essa è unica.

Teorema 1 (Teorema di Church-Rosser)**.** *Se* $M \longrightarrow^\star N_1$ *e* $M \longrightarrow^\star N_2$*, allora esiste* M' *tale che* $N_1 \longrightarrow^\star M'$ *e* $N_2 \longrightarrow^\star M'$*.*

La Figura 1 rappresenta graficamente questo teorema in generale e la Figura 2 rappresenta la sua applicazione al lambda termine $(\lambda x.\lambda y.y)((\lambda z.zz)(\lambda t.tt))$.

L'unicità della forma normale permette di considerare la forma normale a cui si riduce un lambda termine come il suo valore. È, quindi, naturale cercare strategie di riduzione che siano *normalizzanti*, cioè producano la forma normale, se esiste. La strategia per nome è normalizzante (cfr. Barendregt (1984) (Teorema 13.2.2)), mentre quella per valore non lo è, come evidente dall'ultimo esempio. Vi sono molte strategie di riduzione normalizzanti: particolare importanza ha la valutazione *pigra* (in inglese "lazy"), in cui un termine viene ridotto solo quando è utilizzato (cfr. Wadsworth (1971)). Un'ulteriore richiesta per le strategie di riduzione è quella di essere *ottimali*, cioè di usare il minimo numero di passi di riduzione. La strategia per nome non è ottimale, applicata a $(\lambda x.xx)((\lambda y.y)(\lambda z.z))$ usa 4 passi, mentre la strategia per valore applicata allo stesso termine usa 3 passi. In realtà non esiste un modo effettivo di definire una strategia ottimale (cfr. Barendregt (1984) (Teorema 13.5.7)).

Lambda termini interessanti Elenchiamo alcuni lambda termini (usando per loro nomi classici in letteratura) che dimostrano come il lambda calcolo ben si presti alla rappresentazione della manipolazione di funzioni. Per esempi più fantasiosi si consiglia il volume Smullyan (1985).

Il lambda termine

$$B = \lambda x.\lambda y.\lambda z.x(yz)$$

rappresenta l'applicazione della composizione di due funzioni (F_1, F_2) ad un argomento (A):

$$\begin{aligned} BF_1F_2A \;=\; \underline{(\lambda x.\lambda y.\lambda z.x(yz))F_1}F_2A \;&\longrightarrow \underline{(\lambda y.\lambda z.F_1(yz))F_2}A \\ &\longrightarrow \underline{(\lambda z.F_1(F_2z))A} \\ &\longrightarrow F_1(F_2A) \end{aligned}$$

Il lambda termine

$$C = \lambda x.\lambda y.\lambda z.xzy$$

applica una funzione (F) a due argomenti (A_1, A_2) scambiando l'ordine degli argomenti:

$$\begin{aligned} CFA_1A_2 \;=\; \underline{(\lambda x.\lambda y.\lambda z.xzy)F}A_1A_2 \;&\longrightarrow \underline{(\lambda y.\lambda z.Fzy)A_1}A_2 \\ &\longrightarrow \underline{(\lambda z.FzA_1)A_2} \\ &\longrightarrow FA_2A_1 \end{aligned}$$

Ad esempio, CB è la composizione di funzioni in ordine inverso, cioè CB applicato a due funzioni (F_1, F_2) ed a un argomento (A) applica F_2 al risultato dell'applicazione di F_1 ad A:

$$\begin{aligned} CBF_1F_2A \;=\; \underline{(\lambda x.\lambda y.\lambda z.xzy)B}F_1F_2A \;&\longrightarrow \underline{(\lambda y.\lambda z.Bzy)F_1}F_2A \\ &\longrightarrow \underline{(\lambda z.BzF_1)F_2}A \\ &\longrightarrow \underline{BF_2}F_1A \\ &\longrightarrow^{\star} F_2(F_1A) \end{aligned}$$

Il lambda termine

$$W = \lambda x.\lambda y.xyy$$

applica una funzione (F) a due copie di un argomento (A):

$$\begin{aligned} WFA &= \underline{(\lambda x.\lambda y.xyy)F}A \longrightarrow \underline{(\lambda y.Fyy)A} \\ &\longrightarrow FAA \end{aligned}$$

Il lambda termine

$$\overline{2} = \lambda x.\lambda y.x(xy)$$

applica una funzione (F) al risultato dell'applicazione della stessa funzione ad un argomento (A):

$$\begin{aligned} \overline{2}FA &= \underline{(\lambda x.\lambda y.x(xy))F}A \longrightarrow \underline{(\lambda y.F(Fy))A} \\ &\longrightarrow F(FA) \end{aligned}$$

Più complesso ma fondamentale per la costruzione di funzioni è il lambda termine

$$Y = (\lambda x.\lambda y.y(xxy))(\lambda z.\lambda t.z(ttz))$$

chiamato *combinatore del punto fisso di Turing* (cfr. Turing (1937)). Il lambda termine Y applicato ad una funzione F si riduce all'applicazione di F a YF:

$$\begin{aligned} YF &= \underline{(\lambda x.\lambda y.y(xxy))(\lambda z.\lambda t.z(ttz))}F \\ &\longrightarrow \underline{(\lambda y.y((\lambda z.\lambda t.z(ttz))(\lambda u.\lambda v.v(uuv))y))F} \\ &\longrightarrow F((\lambda z.\lambda t.z(ttz))(\lambda u.\lambda v.v(uuv))F) \\ &= F(YF) \end{aligned}$$

Y quindi permette di applicare una funzione un numero arbitrario, anche infinito, di volte.

Se ad esempio applichiamo Y alla funzione $K = \lambda x.\lambda y.x$ che sceglie sempre il primo argomento tra due otteniamo la riduzione infinita:

$$YK \longrightarrow^\star (\lambda x_1.\lambda y_1.x_1)(YK) \longrightarrow \lambda y_1.YK \longrightarrow^\star \lambda y_1.\lambda y_2.YK$$
$$\longrightarrow^\star \lambda y_1 \ldots \lambda y_n.YK \longrightarrow^\star \ldots$$

Quindi YK applicato and un numero arbitrario di argomenti $A_1, A_2, \ldots, A_n$ li "mangia" tutti e si riduce sempre a se stesso:

$$YKA_1A_2 \ldots A_n \longrightarrow^\star YKA_2 \ldots A_n \longrightarrow^\star YKA_n \longrightarrow^\star YK$$

Per questo motivo YK è chiamato "orco".

3. CALCOLABILITÀ

Come già detto nella Sezione 2. i numeri naturali e le operazioni su di essi si possono rappresentare nel lambda calcolo. Vi sono varie rappresentazioni (cfr. Böhm (1963)), qui riportiamo la prima e la più nota che è di Alonzo Church (cfr. Church (1936)).

L'idea alla base dei numerali di Church è che il naturale n sia rappresentato da una funzione $\overline{n}$ che applicata ad una funzione x ed ad un argomento y ripeta n volte l'applicazione di x ad y. Ad esempio:

- la rappresentazione del numero 0 deve restituire y e per questo

  $\overline{0} = \lambda x.\lambda y.y$;

- la rappresentazione del numero 1 deve restituire xy e per questo

  $\overline{1} = \lambda x.\lambda y.xy$;

- la rappresentazione del numero 2 deve restituire $x(xy)$ e per questo

  $\overline{2} = \lambda x.\lambda y.x(xy)$.

In generale:

$$\overline{n} = \lambda x.\lambda y.\underbrace{x(\cdots(x}_{n \text{ volte}}\, y)\cdots)$$

Ci sono lambda termini che rappresentano le operazioni sui naturali, ad esempio:

$$\begin{aligned}(\lambda x.\lambda y.\lambda z.\lambda t.xz(yzt))\overline{m}\overline{n} &\longrightarrow^\star \overline{m+n}\\ (\lambda x.\lambda y.\lambda z.x(yz))\overline{m}\overline{n} &\longrightarrow^\star \overline{m\times n}\end{aligned}$$

Più in generale si può dimostrare che se f è una funzione che, applicata a k naturali, restituisce un naturale o è indefinita e se f può essere calcolata in modo meccanico (ad esempio usando la *macchina di Turing* (cfr. Turing (1936))), allora esiste un lambda termine M_f tale che:

- se $f(n_1,\cdots,n_k)$ è definita, $M_f\overline{n_1}\cdots\overline{n_k} \longrightarrow^\star \overline{f(n_1,\cdots,n_k)}$
- se $f(n_1,\cdots,n_k)$ è indefinita, $M_f\overline{n_1}\cdots\overline{n_k}$ non si riduce ad una forma normale.

Alonzo Church ha introdotto il lambda calcolo anche per caratterizzare la nozione di *calcolabilità effettiva* (cfr. Church (1932)). A lui è dovuta la famosa tesi chiamata appunto *Tesi di Church*:

la possibilità di definire funzioni nel lambda calcolo cattura la nozione informale di calcolabilità effettiva.

Certamente la dimostrazione che l'insieme delle funzioni rappresentabili nel lambda calcolo coincide con l'insieme delle funzioni calcolabili con una macchina di Turing costitui-

sce una pietra miliare a supporto della tesi di Church, considerando quanto diversi siano i due formalismi. Il lettore interessato è invitato a consultare il profilo di Alan Turing nel presente portale (cfr. Frixione e Numerico (2013)).

Nel lambda calcolo si possono anche rappresentare le strutture dati definite in modo ricorsivo quali liste ed alberi (cfr. Böhm e Berarducci (1985)).

4. TIPI

I tipi sono ampiamente usati in svariati campi della logica, della matematica, della filosofia e dell'informatica. Si può far risalire la loro nascita al famoso volume "Principia Mathematica" (cfr. Whitehead e Russell (1910–13)) dove i tipi sono introdotti per evitare i paradossi logici, come il paradosso di Russel (cfr. Russell (1902)). In letteratura i tipi hanno vari significati, ma nel contesto del lambda calcolo e della teoria dei linguaggi di programmazione i tipi sono essenzialmente insiemi di termini con simili proprietà computazionali. Mediante i tipi si possono, perciò, classificare i termini ed in questo modo evitare comportamenti non desiderati. Ad esempio, nei sistemi di tipi presentati in questa sezione la valutazione di ogni lambda termine tipato termina (cfr. Hindley e Seldin (2008) (Teorema 15.31)).

Per quanto riguarda il lambda calcolo i sistemi di tipi possono essere di due generi (cfr. Barendregt (1992)):

- *sistemi di tipi alla Church* in cui i lambda termini sono definiti insieme ai loro tipi;
- *sistemi di tipi alla Curry* in cui i tipi sono assegnati ai termini del lambda calcolo puro da regole formali.

Ci sono molti sistemi di tipi per il lambda calcolo (cfr. Barendregt (1992)), qui ci limiteremo a definire la versione *semplice* dei sistemi di tipi alla Church ed alla Curry. In entrambi questi sistemi i tipi sono costruiti a partire da un insieme infinito enumerabile di variabili di tipo $\phi, \psi, \phi_1, \psi_1, \phi_2, \psi_2, \cdots$ usando unicamente l'operatore $\rightarrow$. Formalmente:

- ogni variabile di tipo è un tipo;
- se σ e τ sono tipi, allora $(\sigma \rightarrow \tau)$ è un tipo.

Il tipo $(\sigma \rightarrow \tau)$ è il tipo di una funzione che applicata ad un argomento di tipo σ produce un risultato di tipo τ.

Per risparmiare parentesi si assume che l'operatore $\rightarrow$ associ a destra, ad esempio $\sigma \rightarrow \tau \rightarrow \rho$ sta per $(\sigma \rightarrow (\tau \rightarrow \rho))$.

Esempi di tipi sono: $\phi \rightarrow \phi$, $\phi \rightarrow \psi \rightarrow \phi$, $(\phi \rightarrow \psi) \rightarrow \phi \rightarrow \psi$, $((\phi \rightarrow \phi) \rightarrow \psi) \rightarrow \psi$.

Nel lambda calcolo tipato semplice alla Church (cfr. Church (1940)) ogni variabile ha un unico tipo ed i termini vengono costruiti insieme ai loro tipi, seguendo lo schema della Definizione 1.

Definizione 4 (Lambda Termini Tipati)**.** *I termini del lambda calcolo tipato semplice sono costruiti come segue:*

- *la variabile x^σ è un lambda termine tipato;*
- *se $M^{\sigma\rightarrow\tau}$ e N^σ sono lambda termini tipati, allora anche la loro applicazione $(M^{\sigma\rightarrow\tau}N^\sigma)^\tau$ è un lambda termine tipato;*
- *se M^τ è un lambda termine tipato, allora la lambda astrazione $(\lambda x^\sigma.M^\tau)^{\sigma\rightarrow\tau}$ è un lambda termine tipato.*

Ad esempio, $(\lambda x^{\sigma}.x^{\sigma})^{\sigma\to\sigma}$ è un lambda termine tipato che rappresenta l'identità su argomenti di tipo σ. Nel calcolo tipato non esiste un'identità universale che possa essere applicata a qualsiasi lambda termine come il termine $\lambda x.x$ del lambda calcolo puro. Altri esempi di termini tipati sono:

$$(\lambda x^{\sigma\to\sigma\to\tau}.(\lambda y^{\sigma}.((x^{\sigma\to\sigma\to\tau}y^{\sigma})^{\sigma\to\tau}y^{\sigma})^{\tau})^{\sigma\to\tau})^{(\sigma\to\sigma\to\tau)\to\sigma\to\tau}$$

$$(\lambda x^{\sigma\to\tau}.(\lambda y^{\rho\to\sigma}.(\lambda z^{\rho}.(x^{\sigma\to\tau}(y^{\rho\to\sigma}z^{\rho})^{\sigma})^{\tau})^{\rho\to\tau})^{(\rho\to\sigma)\to\rho\to\tau})^{(\sigma\to\tau)\to(\rho\to\sigma)\to\rho\to\tau}$$

Per discutere le relazioni fra il lambda calcolo tipato e quello puro è utile definire la funzione $|\ |$ che cancella i tipi:

$$\begin{aligned} |x^{\sigma}| &= x \\ |(M^{\sigma\to\tau}N^{\sigma})^{\tau}| &= |M^{\sigma\to\tau}||N^{\sigma}| \\ |(\lambda x^{\sigma}.M^{\tau})^{\sigma\to\tau})| &= \lambda x.|M^{\tau}| \end{aligned}$$

Ad esempio:

$$|(\lambda x^{\sigma\to\sigma\to\tau}.(\lambda y^{\sigma}.((x^{\sigma\to\sigma\to\tau}y^{\sigma})^{\sigma\to\tau}y^{\sigma})^{\tau})^{\sigma\to\tau})^{(\sigma\to\sigma\to\tau)\to\sigma\to\tau}| = \lambda x.\lambda y.xyy$$

Chiaramente, se M^{σ} è un termine tipato, $|M^{\sigma}|$ è sempre un termine del lambda calcolo puro. In generale, però, dato un termine del lambda calcolo puro non è possibile trovare un termine che lo produca cancellando i tipi. Un esempio paradigmatico è $\lambda x.xx$: la variabile x dovrebbe avere contemporaneamente tipi della forma $\sigma\to\tau$ e σ. Infatti in xx l'occorrenza più a sinistra di x è una funzione il cui argomento è l'occorrenza più a destra

di x.

Nel sistema di tipi semplici alla Curry (cfr. Curry (1934)) ai lambda termini del lambda calcolo puro vengono assegnati tipi a partire da una *base* che è un insieme finito di associazioni *variabile : tipo* in cui tutte le variabili sono distinte. Una base è quindi un insieme $\{x_1 : \sigma_1, \cdots, x_n : \sigma_n\}$ dove $n \geq 0$ e $x_i \neq x_j$ per $1 \leq i \neq j \leq n$: se $n = 0$ la base è l'insieme vuoto.

Le regole di assegnazione di tipi corrispondono alle regole di definizione dei termini del lambda calcolo puro (Definizione 1) e del lambda calcolo tipato (Definizione 4). Scriviamo $\Gamma \vdash M : \sigma$ per dire che dalla base Γ possiamo derivare il tipo σ per il lambda termine M.

Definizione 5 (Sistema di Assegnazione di Tipi)**.** *Le regole di assegnazione di tipi sono:*

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}\,(Ax) \qquad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}\,(\to\ E)$$

$$\frac{\Gamma \cup \{x : \sigma\} \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau}\,(\to\ I)$$

La regola di tipo (Ax) permette di dare ad una variabile il tipo che per quella variabile è assunto nella base. Le regole $(\to\ E)$ e $(\to\ I)$ permettono di dare tipo all'applicazione ed alla lambda astrazione, rispettivamente. Si noti che la regola $(\to\ I)$ elimina la premessa $x : \sigma$ dalla base usata per dare tipo a M. La Figura 3 mostra derivazioni di tipi per i lambda termini $\lambda x.x$, $\lambda x \lambda y.xyy$ e $\lambda x.\lambda y.\lambda z.x(yz)$.

Dalle Definizioni 4 e 5 risulta chiara la relazione fra i tipi semplici alla Church ed alla Curry. Il teorema seguente la formalizza.

$$\dfrac{\dfrac{}{\{x:\sigma\}\vdash x:\sigma}\,(Ax)}{\{\}\vdash \lambda x.x:\sigma\to\sigma}\,(\to\ I)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_1\vdash x:\sigma\to\sigma\to\tau}\,(Ax)\quad\dfrac{}{\Gamma_1\vdash y:\sigma}\,(Ax)}{\Gamma_1\vdash xy:\sigma\to\tau}\,(\to\ E)\quad\dfrac{}{\Gamma_1\vdash y:\sigma}\,(Ax)}{\Gamma_1\vdash xyy:\tau}\,(\to\ E)}{\{x:\sigma\to\sigma\to\tau\}\vdash \lambda y.xyy:\sigma\to\tau}\,(\to\ I)}{\{\}\vdash \lambda x.\lambda y.xyy:(\sigma\to\sigma\to\tau)\to\sigma\to\tau}\,(\to\ I)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_2\vdash x:\sigma\to\tau}\,(Ax)\quad\dfrac{\dfrac{}{\Gamma_2\vdash y:\rho\to\sigma}\,(Ax)\quad\dfrac{}{\Gamma_2\vdash z:\rho}\,(Ax)}{\Gamma_2\vdash yz:\sigma}\,(\to\ E)}{\Gamma_2\vdash x(yz):\tau}\,(\to\ E)}{\{x:\sigma\to\tau,y:\rho\to\sigma\}\vdash \lambda z.x(yz):\rho\to\tau}\,(\to\ I)}{\{x:\sigma\to\tau\}\vdash \lambda y.\lambda z.x(yz):(\rho\to\sigma)\to\rho\to\tau}\,(\to\ I)}{\{\}\vdash \lambda x.\lambda y.\lambda z.x(yz):(\sigma\to\tau)\to(\rho\to\sigma)\to\rho\to\tau}\,(\to\ I)$$

Figura 3: Derivazioni di tipo dove:
$\Gamma_1=\{x:\sigma\to\sigma\to\tau,y:\sigma\}$ e $\Gamma_2=\{x:\sigma\to\tau,y:\rho\to\sigma,z:\rho\}$.

Teorema 2 (Relazione fra Sistemi di Tipi Semplici alla Church ed alla Curry)**.** *Il termine M^σ è un termine del lambda calcolo tipato semplice alla Church se e solo se nel sistema di assegnazione di tipi semplici alla Curry si può derivare*

$$\{x_1:\tau_1,\cdots,x_n:\tau_n\}\vdash |M^\sigma|:\sigma$$

dove $x_1^{\tau_1},\cdots,x_n^{\tau_n}$ sono tutte e sole le variabili che occorrono libere in M^σ.

È facile vedere che, leggendo i tipi come formule logiche, cancellando i lambda termini dalle regole del sistema di assegnazione di tipi semplici alla Curry, si ottengono le regole

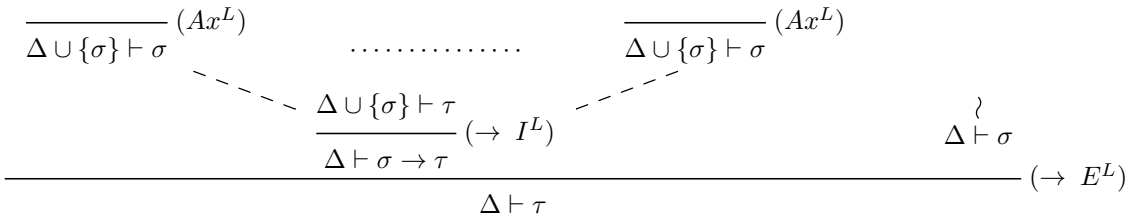


Figura 4: Derivazione logica con taglio.

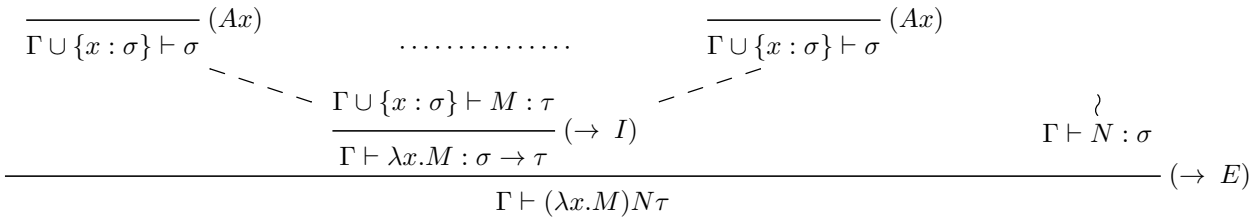


Figura 5: Derivazione di tipo per un β-redesso.

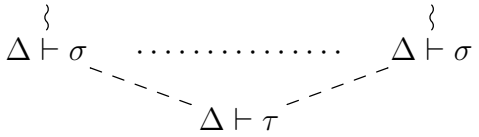
della logica intuizionista con l'operatore di implicazione:

$$\frac{\sigma \in \Delta}{\Delta \vdash \sigma}\,(Ax^L) \qquad \frac{\Delta \vdash \sigma \to \tau \quad \Delta \vdash \sigma}{\Delta \vdash \tau}\,(\to\ E^L)$$

$$\frac{\Delta \cup \{\sigma\} \vdash \tau}{\Delta \vdash \sigma \to \tau}\,(\to\ I^L)$$

dove con Δ indichiamo un insieme di assunzioni $\{\sigma_1, \cdots, \sigma_n\}$.

L'osservazione più interessante è che l'eliminazione del taglio nelle dimostrazioni logiche che trasforma una derivazione della forma mostrata in Figura 4 nella derivazione:



corrisponde alla trasformazione della derivazione di tipo dovuta all'applicazione della β-regola. Infatti la derivazione di tipo in Figura 5 diventa:

$$\begin{array}{ccc} \overset{\wr}{\Gamma \vdash N : \sigma} & \cdots\cdots\cdots\cdots & \overset{\wr}{\Gamma \vdash N : \sigma} \\ & \Gamma \vdash M[x := N] : \tau & \end{array}$$

Questa è l'essenza dell'*isomorfismo di Curry-Howard* (cfr. Curry (1934), Howard (1980)) che si può riassumere nelle seguenti corrispondenze:

tipi	$\Leftrightarrow$	formule logiche
lambda termini	$\Leftrightarrow$	dimostrazioni
β-riduzione	$\Leftrightarrow$	eliminazione del taglio

In altre parole un termine del lambda calcolo corrisponde ad una dimostrazione di una formula logica e la β-riduzione dei lambda termini corrisponde alla semplificazione delle dimostrazioni logiche mediante l'eliminazione del taglio. È interessante osservare che l'isomorfismo di Curry-Howard vale per una grande varietà di sistemi di tipi e di logiche. Questa corrispondenza ha influenzato lo sviluppo sia di logiche che di sistemi di tipi, anche nell'ambito dell'informatica. I linguaggi di programmazione *esplicitamente tipati*, in cui il programmatore deve scrivere il tipo di ogni variabile usata, hanno sistemi di tipi alla Church. Invece, i linguaggi di programmazione *con inferenza di tipo*, in cui il programmatore non deve scrivere tipi, hanno sistemi di tipi alla Curry.

5. LAMBDA CALCOLO E INFORMATICA

Disegno dei linguaggi di programmazione

L'influenza del lambda calcolo sull'informatica è ampia e profonda per molti aspetti. Possiamo affermare che il lambda calcolo sia stato il primo linguaggio di programmazione, precedente persino al alla nozione stessa di calcolatore (in inglese "computer") in senso moderno. Esso è un calcolo molto semplice, potente ed elegante, che è diventato modello formale e ispiratore di quella categoria di linguaggi che rientrano nel paradigma funzionale, in cui la computazione consiste essenzialmente in definizione e applicazione di funzioni. Analogamente le teorie di tipo del lambda calcolo hanno ispirato i sistemi di tipo di diversi linguaggi di programmazione, non solo funzionali ma anche orientati agli oggetti.

Il primo linguaggio funzionale, il *LISP* (che sta per LISt Processing), è stato concepito dal suo ideatore John McCarthy proprio ispirandosi *direttamente* alla lambda notazione di Church (cfr. McCarthy (1960)). L'implementazione originale della β-regola non usa la convenzione di Barendregt e per questo motivo durante le riduzioni occorrenze libere di variabili possono diventare legate (un fenomeno noto come "la cattura delle variabili libere"). Questo non accade nei dialetti del LISP più recenti, come lo Scheme (cfr. Abelson e Sussman (1996)), il Common LISP (cfr. Seibel (2005)), etc. Il LISP ed i suoi dialetti sono ampiamente utilizzati in programmi per l'intelligenza artificiale, lo sviluppo del Web, la finanza, l'insegnamento dell'informatica e molte altre applicazioni. Tutte le più note versioni del linguaggio LISP non prevedono tipi.

Robin Milner ha proposto il primo linguaggio funzionale in cui l'utente può scrivere pro-

grammi senza tipi ma il compilatore inferisce i tipi o manda un messaggio di errore (cfr. Milner (1978)). Il nome di questo linguaggio è *ML*, che significa Meta-Language. Anche in questo caso sono stati sviluppati diversi dialetti: SML (Standard ML) (cfr. Paulson (1996)), CAML (Categorical Abstract Machine Language) (cfr. Cousineau e Mauny (1998)), F# (cfr. Hansen e Rischel (2013)). Le applicazioni di ML includono il disegno e la manipolazione di linguaggi (interpreti, analizzatori, dimostratori automatici di problemi), bioinformatica, sistemi finanziari etc.

Il linguaggio *Haskell* (il cui nome è un omaggio ad Haskell Curry) ha un sofisticato sistema di inferenza di tipi ed usa la strategia di valutazione "pigra", che permette di trattare anche strutture dati infinite, come ad esempio la lista dei numeri primi (cfr. Peyton Jones (2003)). In Haskell si possono scrivere facilmente programmi con valutazioni parallele che sfruttano le capacità delle più recenti architetture degli elaboratori elettronici. Haskell ha una vasta gamma di impieghi commerciali, nell'aerospazio, nella difesa, nella finanza, nella costruzione di start-ups per il Web e nelle società di progettazione hardware.

Oggi le *lambda espressioni* sono diventate un ingrediente fondamentale nei linguaggi di programmazione sia multi-paradigma, come *Python* (cfr. Lutz (2013)), che orientati agli oggetti. Per esempio, la loro recente introduzione in *C++* (cfr. Kalb e Ažman (2015)) e *Java 8* (cfr. Warburton (2014)) ha aggiunto al paradigma di programmazione ad oggetti nuovi modelli di strutturazione del codice, basati sull'uso di funzioni anonime come veri e propri oggetti che possono essere passati a metodi per comporre pezzi di codice, ottenendo così un comportamento più dinamico di quanto fosse possibile senza l'uso delle lambda espressioni. Notiamo che, sebbene linguaggi come C++ e Java siano esplicitamente tipati,

le lambda espressioni sono state introdotte in essi con tipaggio alla Curry, ossia non è necessario che il programmatore specifichi il tipo della variabile legata dal lambda.

Programmazione certificata e dimostratori interattivi

La programmazione certificata si basa sull'isomorfismo di Curry-Howard. L'idea è che sviluppare un programma che soddisfi una specifica equivale a trovare una dimostrazione di una formula logica. In questo modo il programma ottenuto è accompagnato da una dimostrazione della sua correttezza (cfr. de Bruijn (1970), Martin-Löf (1984)).

Il Coq (cfr. Huet, Kahn, e Paulin-Mohring (2006)) è un dimostratore interattivo che si basa su un lambda calcolo con tipi dipendenti e quantificazione universale delle variabili (cfr. Coquand e Huet (1988)). Usando Coq si possono gestire asserzioni matematiche e controllare meccanicamente le dimostrazioni di queste asserzioni. Inoltre Coq aiuta nella ricerca di dimostrazioni formali costruttive ed estrae programmi dalle dimostrazioni costruttive delle loro specifiche formali.

Vi sono molti altri dimostratori interattivi basati su lambda calcoli ed implementati in linguaggi funzionali: ricordiamo NuPRL (cfr. Constable et al. (1986)), Isabelle (cfr. Paulson (1989)), LEGO (cfr. Luo e Pollack (1992)).

Semantica dei linguaggi di programmazione

La prima semantica formale di un linguaggio di programmazione reale è stata data da Peter Landin che ha tradotto il nucleo centrale del linguaggio Algol 60 (cfr. Backus et al. (1963)) in un'estensione del lambda calcolo (cfr. Landin (1965)).

Dana Scott ha sviluppato la teoria dei domini per modellare il lambda calcolo (cfr. Scott (1972)). Questa teoria è la base della semantica denotazionale dei linguaggi di programmazione (cfr. Stoy (1977)).

Il lambda calcolo degli oggetti è un'estensione del lambda calcolo che è stata usata per studiare i linguaggi di programmazione orientati agli oggetti, con particolare riguardo ai meccanismi di tipaggio (cfr. Abadi e Cardelli (1996)).

6. VARIE

Testi

Il libro Hindley e Seldin (2008) è un'introduzione al lambda calcolo senza e con tipi. Volumi con trattazioni più estese sono Barendregt (1984) per il lambda calcolo puro e Barendregt, Dekkers, e Statman (2013) per il lambda calcolo con tipi. Un'esaustiva visione storica del λ-calcolo e della Logica Combinatoria si trova nell'articolo Cardone e Hindley (2009).

Serie di conferenze

Il lambda calcolo è a tutt'oggi un interessante argomento di ricerca: molti lavori sul lambda calcolo vengono presentati a conferenze internazionali altamente selettive e pubblicati su prestigiose riviste. In particolare l'annuale conferenza *FSCD* (Formal Structures for Computation and Deduction, si veda `http://www.cs.ox.ac.uk/conferences/fscd2017/about.html`) è in gran parte dedicata a nuovi risultati incentrati sul lambda calcolo.

Ringraziamenti

Corrado Böhm è stato per Mariangiola e per molti ricercatori una fantastica guida nel mondo dell'informatica teorica ed in particolare del λ-calcolo. Lui ha saputo trasmettere la sua immensa curiosità ed il suo amore senza limiti per la ricerca.

La presente versione di queste note è migliorata grazie ai suggerimenti dei revisori anonimi.

BIBLIOGRAFIA

Abadi, M., Cardelli, L. (1996). *A theory of objects*. Springer-Verlag.

Abelson, H., Sussman, G. J. (1996). *Structure and interpretation of computer programs*. M.I.T. Press.

Backus, J., Bauer, F., Green, J., Katz, C., McCarthy, J., Naur, P., ... Woodger, M. (1963). *Revised report on the algorithmic language Algol 60*. IFIP. (http://www.masswerk.at/algol60/report.htm)

Barendregt, H. P. (1984). *The lambda calculus: its syntax and semantics*. North-Holland.

Barendregt, H. P. (1992). "Lambda calculi with types." In *Handbook of logic in computer science, volume 2: Background: Computational structures* (pp. 117–309). Oxford University Press.

Barendregt, H. P., Dekkers, W., Statman, R. (2013). *Lambda calculus with types*. Perspectives in logic. Cambridge University Press.

Böhm, C. (1963). "The CUCH as a formal and description language." In *Annual review in automatic programming, vol. 3* (pp. 179–197). Pergamon Press.

Böhm, C., Berarducci, A. (1985). "Automatic synthesis of typed Λ-programs on term algebras." *Theoretical Computer Science*, *39*, 135–154.

de Bruijn, N. G. (1970). "The mathematical language AUTOMATH, its usage and some of its extensions." In *Symposium on automatic demonstration.* Lecture Notes in Mathematics (Vol. 125, pp. 29–61). Springer-Verlag.

Cardone, F., Hindley, J. R. (2009). "Lambda-calculus and combinators in the 20th century." In *Logic from Russell to Church.* Handbook of the History of Logic (Vol. 5, pp. 723–817). Elsevier.

Church, A. (1932). "A set of postulates for the foundation of logic (1)." *Annals of Mathematics*, *33*, 346–366.

Church, A. (1936). "An unsolvable problem of elementary number theory." *American Journal of Mathematics*, *58*, 354–363.

Church, A. (1940). "A formulation of the simple theory of types." *The Journal of Symbolic Logic*, *5*, 56–68.

Constable, R., Allen, S. F., Bromley, H. M., Cleaveland, W. R., Cremer, J. F., Harper, R. W., ... Smith, S. F. (1986). *Implementing mathematics with the NuPRL proof development system*. Prentice-Hall.

Coquand, T., Huet, G. (1988). "The calculus of constructions." *Information and Computation*, *76*, 95–120.

Cousineau, G., Mauny, M. (1998). *The functional approach to programming*. Cambridge University Press.

Curry, H. B. (1930). "Grundlagen der kombinatorischen Logik." *American Journal of*

Mathematics, *52*, 509–536, 789–834.

Curry, H. B. (1934). "Functionality in combinatory logic." *Proceedings of the National Academy of Science of the USA*, *20*, 584–590.

Frixione, M., Numerico, T. (2013). "Profili: Alan Mathison Turing." *www.aphex.it*, *7*, 511–562.

Hansen, M. R., Rischel, H. (2013). *Functional programming using F#*. Cambridge University Press.

Heijenoort, J. v. (Ed.). (1967). *From Frege to Gödel: a source book in mathematical logic, 1879 –1931*. Harvard University Press.

Hindley, J. R., Seldin, J. P. (2008). *Lambda-calculus and combinators, an introduction*. Cambridge University Press.

Howard, W. (1980). "The formulas-as-types notion of construction." In *To H. B. Curry: Essays on combinatory logic, lambda calculus and formalism* (pp. 479–490). Academic Press.

Huet, G., Kahn, G., Paulin-Mohring, C. (2006). *The Coq proof assistant, a tutorial*. INRIA. (http://coq.inria.fr/V8.1/tutorial.html)

Kalb, J., Ažman, G. (2015). *C++ today: The beast is back*. O'Reilly.

Landin, P. J. (1965). "A correspondence between ALGOL 60 and Church's lambda notation." *Communications of the ACM*, *8*, 89–101,158–165.

Luo, Z., Pollack, R. (1992). *The LEGO proof development system: A user's manual* (Tech. Rep. No. ECS-LFCS-92-211). University of Edinburgh: Department of Computer Science.

Lutz, M. (2013). *Learning Python*. O'Reilly.

Martin-Löf, P. (1984). *Intuitionistic type theory*. Studies in Proof Theory. Bibliopolis.

McCarthy, J. (1960). "Recursive functions of symbolic expressions and their computation by machine." *Communications of the ACM*, *3*, 184–195.

Milner, R. (1978). "A theory of type polymorphism in programming." *Journal of Computer and System Sciences*, *17*, 348–375.

Paulson, L. C. (1989). "The foundation of a generic theorem prover." *Journal of Automated Reasoning*, *5*, 363–397.

Paulson, L. C. (1996). *ML for the working programmer*. Cambridge University Press.

Peano, G. (1889). *Arithmetices principia, nova methodo exposita*. Bocca. (Traduzione in (Heijenoort, 1967), pages 83–97)

Peyton Jones, S. L. (2003). *Haskell 98 language and libraries: the revised report*. Cambridge University Press.

Russell, B. (1902). *Letter to Frege.* Traduzione in (Heijenoort, 1967), pages 124–125.

Schönfinkel, M. (1924). "Über die Bausteine der mathematischen Logik." *Mathematische Annalen*, *92*. (Traduzione in (Heijenoort, 1967), pages 355–366)

Scott, D. S. (1972). "Continuous lattices." In *Toposes, algebraic geometry and logic.* Lecture Notes in Mathematics (Vol. 274, pp. 97–136). Springer-Verlag.

Seibel, P. (2005). *Practical common Lisp*. Apress. (http://www.gigamonkeys.com/book/)

Smullyan, R. (1985). *To mock a mockingbird*. A. A. Knopf.

Stoy, J. E. (1977). *Denotational semantics: The Scott-Strachey approach to programming language semantics*. M.I.T. Press.

Turing, A. (1936). "On computable numbers with an application to the entscheidungs-problem." *Proceedings London Mathematical Society (2)*, *42*, 230–265.

Turing, A. (1937). "The $\mathfrak{p}$-function in λ-K-conversion." *Journal of Symbolic Logic*, *2*, 164.

Wadsworth, C. P. (1971). *Semantics and pragmatics of the lambda-calculus* (Tesi di Dottorato). University of Oxford.

Warburton, R. (2014). *Java 8 Lambdas: Functional programming for the masses*. O'Reilly.

Whitehead, A. N., Russell, B. A. W. (1910–13). *Principia mathematica*. Cambridge University Press.